An Architecture for Highly Available Wide-Area Service Composition

Bhaskaran Raman **Randy H. Katz**

475 Soda Hall, EECS Department, U.C.Berkeley
Berkeley, CA 94720-1776, U.S.A.
{bhaskar,randy}@cs.berkeley.edu
Ph: +1-510-642-8284, Fax: +1-510-643-7352

Abstract

Service composition provides a flexible way to quickly enable new application functionalities in next generation networks. We focus on the scenario where next generation portal providers "compose" the component services of other providers. We have developed an architecture based on an overlay network of service clusters to provide failure-resilient composition of services across the wide-area Internet: our algorithms detect and recover quickly from failures in composed client sessions.

In this paper, we present an evaluation of our architecture whose overarching goal is quick recovery of client sessions. The evaluation of an Internet-scale system like ours is challenging. Simulations do not capture true workload conditions and Internet-wide deployments are often infeasible. We have developed an emulation platform for our evaluation – one that allows a realistic and controlled design study. Our experiments show that the control overhead involved in implementing our recovery mechanism is minimal in terms of network as well as processor resources; minimal additional provisioning is required for this. Failure recovery can be effected using alternate service replicas within about 1 sec after failure detection on Internet paths. We collect trace data to show that failure detection itself can be tight on wide-area Internet paths – within about 1.8 sec. Failure detection and recovery within these time bounds represents a significant improvement over existing Internet path recovery mechanisms that take several tens of seconds to a few minutes.

Keywords: Service composition, Service cluster, Overlay networks, Failure detection, Session recovery, Network emulation

1 Introduction

Value added services and content provisioning will be the driving force behind the development and deployment of future communication networks. It is important to enable quick and flexible development of end application functionality. Composition of services from independent components offers a flexible way to enable new application functionalities. Consider for instance a user with a new wireless thin client roaming to a foreign network. She wishes to access a local news/weather video service. A portal provider enables this by composing the video service with an appropriate transcoder to adapt the contents of the video to the thin client's capabilities (Fig. 1). Further, she wishes to access her email from her home provider on her cell-phone while she is on the move. The portal provider enables this by composing a third-party text-to-speech conversion engine, with the user's email repository. In either example, a novel composite service functionality is enabled through the composition of existing service components. We term the set of component services that are strung together as a *service-level path*.

Composition of complex services from primitive components enables quick development of new application functionality through the reuse of the components for multiple compositions. We envision a wide variety of service components such as media transcoding agents (audio/video), rate-adapting agents, media transformation engines (e.g. text-to-speech), redirection/filtering agents, personalization/customization/user-interface agents, etc. These would be deployed and managed by a variety of service providers and be available for composition of new applications for novel devices in next-generation networks.

Composition by itself is not a novel idea. However, there are critical challenges to be addressed in the context of composing independent components across multiple service providers. When providers deploy service instances independently, the composed service-level path could span multiple Internet domains, as shown in Fig. 1. This has implications on the *availability* of the composed service. Service providers would like to have their services always available for client sessions. However, studies have shown that inter-domain Internet path availability is very poor [1], and that Internet route recovery can take of the order of a few minutes [2]. This in turn reflects on the availability of the composed service. Since multimedia sessions could last for several minutes to hours, it is important to address network failures *during* a session, to improve the overall availability.

Our approach to address this issue involves the use of alternate service replicas as well as alternate Internet paths when the original service-level path experiences an outage. The dotted lines in Fig. 1 show one such alternate service-level path for the composed video session. It is important to ensure that such recovery is *quick* for real-time applications. In taking such an approach to recovery, we assume that services have only soft-state, and no persistent

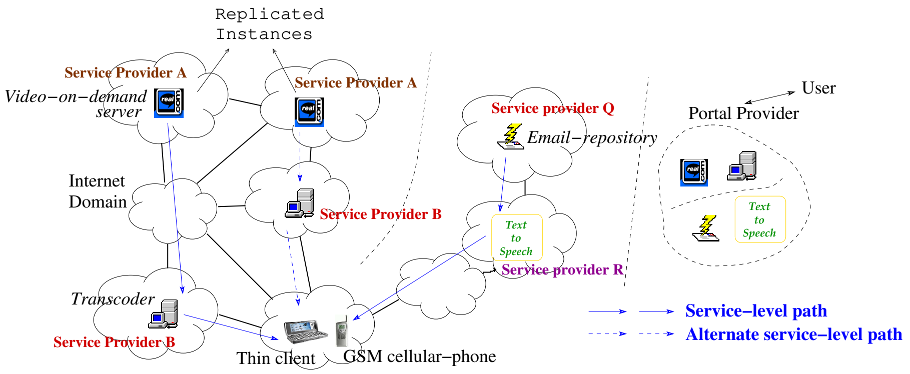


Figure 1. Service composition across the wide-area Internet

state. Soft-state can be built up at an alternate server without affecting the correctness of the session. Our examples fall under this category (also see [3]). Quick restoration of service-level paths is challenging since there are *scaling* implications when a large number of client sessions have to be restored on a failure.

A second challenge is that of *performance* of the composed service through appropriate choice of service instances for the composition. This is more challenging than traditional web-server selection since we have to choose a *set* of service instances, ensuring network reachability and performance along the entire path.

A third issue is that of *failure detection* over the wide-area Internet. There is inherent variability in delay, loss-rates, and outage durations on an inter-domain Internet path. A conservative timeout mechanism to detect failures could mean longer detection times in general, while a more aggressive mechanism may trigger *spurious* path restorations.

We have developed an architecture for addressing the issues of availability and performance in service-level paths. In this paper, we present an evaluation of our design. We specifically look at the issue of quick failure detection and recovery for availability of the service-level path. A challenge that relates to evaluation of mechanisms for path recovery is the following. Simulations are not ideal for capturing true processor/network overheads, especially under scale. However, creating and maintaining a realistic research testbed across the wide-area Internet would be too cumbersome and expensive. Also, with a real deployment, a controlled design study (to identify system bottlenecks) would be difficult due to non-repeatability of experimental conditions.

Our evaluation is based on an emulation platform that we have developed. The platform allows us to realistically implement our algorithms, while emulating wide-area Internet latency and loss. The different instances of our distributed recovery algorithm run on multiple machines of a cluster within our testbed.

Our experiments show that the control overhead involved in updating distributed state to effect service-level path restoration is manageable, both in terms of network resources as well as processor resources. This allows the system to scale well with an increasing number of simultaneous client sessions. In our implementation, a single machine (Pentium-III 500MHz) can easily handle the path state associated with about 400-500 simultaneous client sessions. (Beyond this, we run into bottlenecks in our emulation setup). This amounts to little additional provisioning, especially when dealing with heavy-weight service components such as the video transcoder or the text-to-speech engine of our example (these have much higher provisioning requirements).

To analyze how quickly failure *detection* can be done, we collect trace-data on Internet path outages across geographically distributed hosts. Our analysis shows that failure detection itself, over the wide-area can be done quite aggressively, within about 1.8 sec. We use the traces to model losses and outages on Internet paths and use this to drive our emulation. We find that even with an aggressive failure detection timeout of 1.8 sec, spurious path restorations happen infrequently – about once an hour. Also, the overhead associated with each spurious path restoration is small.

Under our trace-based modeling of Internet outages, we find that *recovery* of paths after failure detection can be done within 1 sec for over 90% of the client sessions. Such a combination of quick detection and recovery, within a few small number of seconds, would be immensely useful for the kinds of real-time composed applications described above.

The rest of the paper is organized as follows. The next section (Section 2) presents an overview of our architecture. Section 3 describes the emulation testbed. Our evaluation of path recovery mechanisms is in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Design Overview

In this section, we present a walk-through of our architecture. We highlight the main design points and establish the context for the performance evaluation. The goal of our architecture is to enable service-level path recovery upon failure detection, and performance-sensitive choice of service instances for path creation as well as recovery. The idea behind path recovery is to use an alternate Internet path, much as in [4, 5], as well as possibly alternate service replicas. The motivation for this is that inter-domain network-path failures can happen quite often – studies show that inter-domain Internet paths can have availability as low as 95% [1]. Importantly, when such failures do happen, they can last for several tens of seconds to several minutes [2].

The choice of service instances for service-level path creation/recovery is somewhat like web-mirror selection, but is more complicated, since in general, we may need to select a *set* of instances for a client session. Further, unlike traditional web-server selection mechanisms, client sessions in our scenario could last for a long time, and it is desirable to provide mechanisms for path recovery using alternate service instances *during* a session.

A hop-by-hop approach where each leg of the path is constructed independently could result in sub-optimal paths – a good choice of the first leg of the path could mean a poor choice for the second leg. Or, it may even happen that no instance of the required second service for composition is reachable from the first chosen service replica. Hence, a simple architecture that uses such a hop-by-hop approach is not appropriate for ensuring availability and performance in wide-area service composition. We reject this approach.

Since service components are central to composition, we think in terms of service-execution platforms, and a service-level *overlay network*. Our architecture for composition is depicted in Fig. 2. We have three planes of operation: at the lowest layer is the hardware platform consisting of compute clusters deployed at different points on the Internet. This constitutes the middle-ware platform on which service providers deploy their services. Providers could deploy their own service cluster platforms, or could use third party providers' clusters. We define a logical overlay network on top of this. At the top-level, service-level paths are constructed as paths in the overlay network. Fig. 3 explains this better.

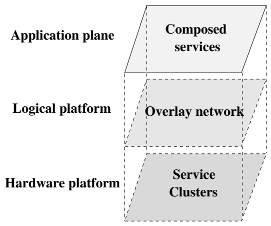


Figure 2. Architecture: Three layers

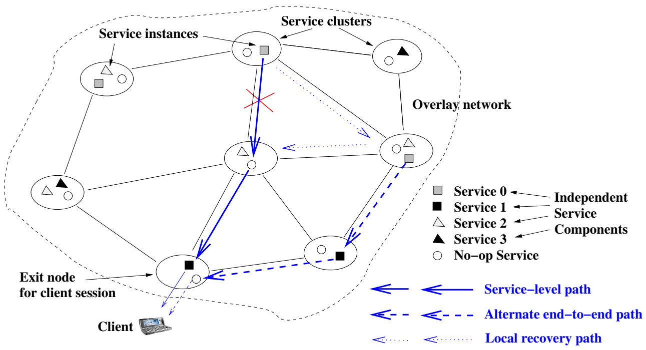


Figure 3. Architecture

Fig. 3 shows the architectural components as they would be deployed on the Internet. Each oval in the figure represents a service cluster execution platform. Each cluster has one or more independent service components. A service-level path is formed as a path in the overlay network. An example is shown in the figure, using an instance each of "Service 0" and "Service 1", in that order. Each cluster also implements a (trivial) "no-op" service that simply provides data connectivity, and does not perform any operation on the data. These no-op services allow composition of services that are not necessarily adjacent in the overlay network.

The overlay network provides the context for exchange of reachability and performance information to create service-level paths. Redundancy in the overlay network allows us to define alternate service-level paths to recover from failures – the dotted lines in Fig. 3.

The use of cluster execution platforms as building blocks is an important design feature. We leverage known mechanisms to handle process and machine level failures of service instances within each cluster execution platform [6]. And we design and evaluate mechanisms to focus on handling wide-area network path failures. Further, with the use of clusters, the overhead of monitoring the liveness of an Internet path representing an overlay link, as well as the overhead of maintaining the distributed overlay graph state, are amortized across all client sessions and all service instances. That is, these overheads are neither dependent on the number of client sessions nor on the number of service instances in deployment.

We now divide the remainder of our discussion in this section into three parts. We first describe the various software functionalities and their interaction to enable composition, in Section 2.1. Then we discuss the important aspect of the scale and extent of the service overlay network, in Section 2.2. In the context of these discussions, Section 2.3 brings out the various aspects of the system that call for quantitative evaluation.

2.1 Software functionalities

For each composed client session, the data exits the overlay network, after passing through the required set of services. The overlay node at which the data exits is called the *exit-node* for that particular client session. The exit node is the one that interfaces with the client, and is responsible for handling client requests for service composition. A client and the associated exit node are shown in Fig. 3.

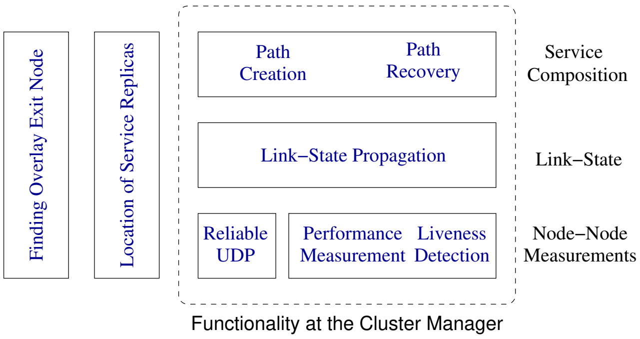


Figure 4. Software Architecture of the Various Functionalities

For a particular client, the choice of the exit overlay node could be made using pre-configuration, or some simple selection mechanism. Fig. 4 shows the various software functionalities in our architecture. The first vertical layer in Fig. 4 captures the functionality of finding an exit node. The next functionality we separate is that of service-location. This is the second vertical layer in Fig. 4. Here, we just need a list of locations of service replicas – something like the list of mirrors for a web-site. This can either be distributed slowly across the overlay nodes, or can be retrieved from a central (replicated) directory.

In each cluster, a *cluster manager (CM)* is responsible for implementing our algorithms for service-level path creation and recovery. The software architecture at the CM is also shown in

Fig. 4. The CM implements the mechanisms for *inter-cluster*, wide-area distributed service-level path creation and recovery.

The functionality at the manager node is in three layers (Fig. 4). The lowest layer implements communication between adjacent nodes (service-clusters) in the overlay network. This includes liveness tracking and performance measurement. We have implemented liveness tracking as a simple periodic two-way heart-beat exchange, with a timeout to signal failure. In this paper, we consider latency as a performance measure – our architecture also allows measurement and exchange of other metrics such as cluster load, bandwidth, or other generic metrics.

At the next layer, global information about overlay link liveness and performance is built using a link-state algorithm in the overlay network. A link-state approach gives global information about the entire overlay graph. This is used in combination with the service-location information to construct service-level paths, at the top layer.

The top layer implements the functionalities for service composition itself: initial creation, and recovery when overlay network failures are detected. The client sends the request for composition to (the cluster-manager of) its chosen exit overlay node. This CM then constructs the service-level path by choosing a particular set of service instances and paths between them in the overlay network. For this, it uses the overlay graph information built up by the link-state layer, as well as the service-location information. On choosing the service-level path, the exit node then sends signaling messages to setup the path.

The messaging at the link-state and service-composition layers are implemented on top of a UDP-based messaging layer that provides at-least-once semantics using re-transmits.

Since all the computations and control messaging relevant to composition are done at the cluster manager of each overlay node, in our discussion below, unless mentioned otherwise, we use the terms "cluster-manager" and "overlay-node" interchangeably – the cluster manager is the one at the overlay cluster node.

The algorithm used for choosing the set of service instances is based on the Dijkstra's algorithm on a transformation of the overlay graph [7]. (The transformation ensures that the path chosen in the graph has the required set of service instances in the required order). We skip the details of this here since it is not relevant for our evaluation. In [8], we have studied how this algorithm can be used in combination with a load balancing metric to balance load across service replicas. However, in this paper, we simply use a latency metric and choose service instances to minimize the end-to-end latency in the service-level path.

When a failure is detected, there are two kinds of recovery mechanisms possible, as in MPLS [9]. We could have end-to-end path recovery or perform local-link recovery. In end-to-end path restoration, the failure information propagates downstream, to the exit-node, which then constructs an alternate service-level path. This construction resembles the original path construction process. In local-link recovery, the failure is corrected locally, by choosing a local path to get around the failed edge. Both kinds of recovery are shown in Fig. 3 using dotted lines. In either case, the composed application session continues in the alternate path, from the point where the old path failed and stopped (we assume client support at the application layer to implement this).

Finally, we make one crucial observation. We note that service-level paths have an explicit session setup phase, and there is connection-state at the intermediate nodes. For instance, for a transcoder service, this switching state includes the input data type and source stream, and the output data type and next-hop destination information. This means that, unlike Internet routing, failure information need not propagate to the entire network and stabilize before corrective measures are taken. This is an important aspect of the system that allows quick restoration of client sessions.

2.2 Scale of the overlay network

In our architecture, an important issue is that of the size and extent of the overlay network. We discuss this now. We first note that the portion of the service-level path after the exit node is not "protected". That is, failures on this portion of the path are neither monitored nor recovered. Hence, ideally, each client should have an exit node "close" to it. It should be close in the sense that the client should experience roughly the same network connectivity to the rest of the Internet as its chosen exit-node. In this sense, the overlay network should span the Internet. The question then is, how many overlay nodes are required to achieve this.

As a point in comparison, we consider the Autonomous-System (AS) network in the Internet. By definition, it spans the Internet since the AS network is what constitutes the Internet. Also, by definition, each node within an AS has roughly the same inter-domain connectivity to the rest of the Internet (like in the definition of "close" in our case in the previous paragraph). The AS network had about 12,000 nodes as of Dec 2001.

Another useful point of comparison for the size of the overlay topology is another Internet-wide service in operation – the Akamai content-distribution network of cache servers (www.akamai.com). While this network is not an "overlay" network in that it does not do routing of user-data, it is similar to our overlay network in that it is an Internet-wide service. Here too, the goal is to span the Internet so that there is a cache server close to each client. This service had an expanse of 13,000+ server locations in 1000+ ISP network locations as of Oct 2001 (www.akamai.com).

As an estimate, using these two points of comparison, we can say that a few thousand nodes are probably sufficient to span the current Internet. Making a stronger claim about the exact number of overlay nodes required is an interesting research issue in itself and is out of scope of this work. However, the ballpark figure of a few thousand nodes (for the overlay size) suffices for the purposes of our evaluation below.

2.3 Potential scaling bottlenecks and sources of overhead

Each of the layers of functionality in Fig. 4 has overheads. There are two different issues of scale that arise.

The first is with respect to the number of simultaneous client sessions. At the service-composition layer, the presence of connection-state per path makes quick failure recovery easier. This is because recovery is per client session, and does not depend on propagation and stabilization of the failure information across the network. However, this could have scaling implications since a large number of client sessions may have to be restored on failure of an overlay link.

The second scaling issue concerns the size of the overlay network. During path creation or restoration, finding a path through a set of intermediate service instances involves a graph computation based on the information collected by the link-state layer. This could have memory or CPU bottlenecks for a large overlay network. Further, the choice of a link-state approach for building global information could pose problems. Link-state flooding consumes network bandwidth, and this could be a potential source of bottleneck for a large network.

Apart from these issues of scale, at the lowest layer of Fig. 4, failure detection itself is a concern when service-clusters adjacent in the overlay network, are separated over the wide-area Internet. A conservative mechanism to detect failures (in the Internet path in-between) could mean longer detection times in general, while a more aggressive mechanism may trigger spurious path restorations. A spurious path restoration is wasted effort and represents a form of system overhead.

Our main goal in the rest of the paper is to identify sources of scaling bottlenecks, quantify the various overheads, and determine how quickly we can effect service-level path recovery. We now turn to describing our evaluation testbed to study these overheads.

3 Experimental testbed

Evaluation of an Internet-scale system like ours is challenging since performance metrics such as time-to-recovery from failure and scaling with the number of clients depend on Internet dynamics. A large-scale wide-area testbed is cumbersome to setup and maintain. Simulations are inappropriate since they do not capture processing bottlenecks. They also do not scale for large numbers of client sessions. We have developed a *network-emulation platform* for our experiments. We have a real implementation of the distributed algorithms and mechanisms. Instead of having them run across the wide-area, we have the different components run on different machines in a cluster connected by a high-speed LAN. We then emulate the wide-area network characteristics between the machines.

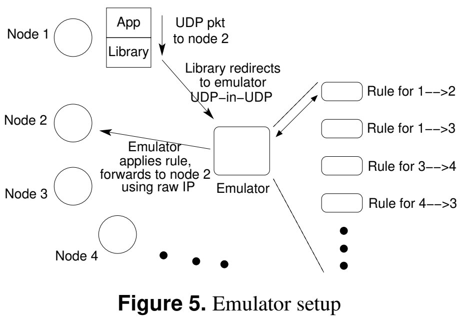


Figure 5. Emulator setup

The opportunity for such an emulation-based platform is provided by the Millennium cluster (www.millennium.berkeley.edu). Our setup is shown in Fig. 5. Each machine represents an overlay node of our architecture. We run only the cluster-manager software of an overlay node on the testbed machine corresponding to it. This is because the CM is the one responsible for all the control traffic associated with service composition. We have all traffic between the distributed components pass through an *emulator* machine. (Note that the millennium cluster in our emulation is quite different from the service-clusters in our architecture. In fact, each node in our emulation setup represents a cluster manager of a service-cluster/overlay-node in our architecture, and runs the software shown in Fig. 4). The emulator has rules to capture the behavior of the overlay link between pairs of overlay nodes. We have modeled delay/latency behavior between overlay nodes, as well as the frequency and duration of failures of the overlay link. The actual settings for these packet handling rules, and the choice of the overlay topology itself, are presented in Sec. 4.1.

Each emulation node in our testbed is a 500MHz Pentium-III machine with up to 3GB memory, and a 500KB cache. Each is a 2-way, or 4-way multi-processor, and runs Linux 2.4. The emulator is setup on a Pentium-4 1500MHz machine with 256MB memory, and 256KB cache, running Linux 2.4.2-2. It is on a 100Mbps network.

Table 1 presents a brief characterization of our emulator setup. We have traffic passing through the emulator at a constant packet rate, with all packets being the same size. The emulator fires a randomly picked rule for each packet. In this setup, we vary the packet rate and packet size across different runs, and measure the percentage of packets lost at the emulator (these losses exclude the packet drops at the emulator as stipulated by the randomly picked rule).

	10,000/sec	15,000/sec	20,000/sec	25,000/sec
250B	0.000	0.020	0.005	23.9
500B	0.010	0.020	0.185	20.4
800B	0.86	8.72	29.24	44.1
1100B	1.63	36.14	49.75	64.71
1400B	36.36	50.65	65.48	68.95

Table 1. % packets lost by the emulator

In Table 1, the scaling limits of the emulator are reached in both dimensions – at large packet sizes and at high packet rates. We note that the emulator performs quite well for up to a packet rate of 20,000 pkts/sec, for pkt sizes below 500 bytes. This constitutes about $20,000 \times 500 \times 8 = 80Mbps$, which is close to the ethernet limit in our setup. We shall refer back to these numbers later to verify that in our experiments, we do not exceed these limits of operation of the emulator.

4 Evaluation

In this section, we turn to the evaluation of our system. We seek to understand the scaling behavior of the system and quantify overheads as summarized in Section 2.3. In our set of experiments, we consider several metrics: (a) the time to *recovery* of client path sessions, after failure detection, (b) the time to *detection* of failures in Internet paths, (c) the additional control overhead due to spurious path restorations, and (d) other memory, CPU, and network overheads in our software architecture. We study client session recovery time as a function of the number of client sessions (load) at each CM. We analyze the two different recovery algorithms presented in Section 2.1. In Sec. 4.2, we consider end-to-end path recovery and study its scaling behavior. In Sec. 4.3, we compare local recovery with end-to-end recovery.. For these set of experiments, we use realistic modeling of Internet delay, but use controlled link failures. We then turn to a trace-based study of

Internet path failure behavior in Sec. 4.4, and look at failure detection. Using this trace data, we study the time to path recovery under Internet failure patterns in Sec. 4.5. This allows us to examine spurious path restorations. Finally, we look at other sources of overhead in our system in Sec. 4.6.

4.1 Parameter settings for the experiments

Before presenting our experiments, we explain two important parameter settings in this subsection: the overlay topology, and the nature of performance variation of the links in the overlay network.

The Overlay Network Topology

While we envision a full-fledged deployment of our architecture to constitute a few thousand overlay nodes (Sec. 2.2), we first wish to study system behavior with a smaller number of nodes. This is also a limitation imposed by our emulation testbed which has a maximum of a hundred machines to act as overlay nodes. However, we study scaling in the dimension of the number of client sessions with this setup.

We use the following procedure to generate an overlay network. We first generate an underlying *physical network* with a Transit-Stub topology. This graph has a total of 6,510 nodes, and 20,649 edges. This topology is generated using the GT-ITM package [10] (with 14 transit ASes, each with 15 nodes, 10 stub-ASes per transit-node, and 3 nodes per stub-AS). We then select a random subset of *N* nodes from this physical network to generate an *N*-node overlay topology (*N* is a much smaller number than 6,510). Next, we examine pairs of overlay nodes in the order of their closeness and decide to form overlay links between these. Overlay links are thus equivalent to *physical paths*. In this process, we impose the constraint that no physical link is shared by two overlay links. (Although this could theoretically result in a disconnected overlay topology, for the graph that we used, the final overlay network was connected).

Overlay Network Parameters

To study our mechanisms for service-level path creation, adaptation, and recovery, we vary two network parameters: latency, and occurrence of failures (packet drops are modeled simply as short failures). We use these two parameters to capture the nature of overlay links in our emulations. Each rule at the emulator involves these two parameters.

Latency Variation: To model this, we use results from a study of round-trip-time (RTT) behavior on the Internet [11]. We make use of two results: (1) Significant changes (defined as over 10 ms) in average RTT, measured over 1 minute intervals occur only once in about 52 min. This value of 52 min is averaged over all host-pairs. (2) The average run length of RTT, within a jitter of 10 ms, is 110 seconds across all host-pairs. The first result says that sustained changes in RTT occur slowly, and the second result says that the jitter value is quite small for periods of the order of 1-2 minutes.

We use these as follows. The costs of edges of the physical network are as generated by the GT-ITM package. For the overlay links, the cost is simply the addition of the costs of the physical path edges between the overlay nodes. This cost is however, only relative. We normalize this by setting the maximum overlay link cost of *100 ms*. This is the one-way cost, and is the base-value for the latency in an overlay link. Given a base-value L for the latency, we gradually vary the latency between L and $2L$. Such a variation of overlay link cost gives a maximum one-way latency of $2 \times 100 = 200\ ms$, and a max RTT of up to $2 \times 200 = 400\ ms$. This is a reasonable choice since overlay links are likely to be formed between "close-by" overlay nodes – they are unlikely to be separated by an RTT of over 400 ms. We impose the constraint that significant sustained changes happen once in an "epoch" of length 52 min (using result (1)). Also, to have some variability, we set a value of 15 min for this epoch for 10% of the overlay links, and 100 min for another 10% (the rest 80% have the value of 52 min). Within an epoch of RTT value, 1 min averages are varied within 10 ms (in accordance with (1)). And within a minute, jitter is within 10 ms (in accordance with (2)).

In our modeling of latency variation, we do not include occasional, isolated RTT spikes that do happen [11]. Instead, we model RTT spikes also as loss-periods/failures, which is worse than RTT spikes. (Although the study we have used is somewhat old, it is extensive. Also, our own UDP-based experiments in Sec. 4.4 agree in spirit with observation (2) above – in our experiments, we observe that outage periods lasting beyond 1-2 sec are very rare).

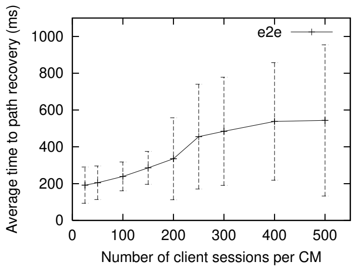
Occurrence of failures: For the initial set of experiments, we fail graph links in a controlled fashion. We then used a trace-based emulation of network failures. We postpone a discussion of this emulation to Sec. 4.4.

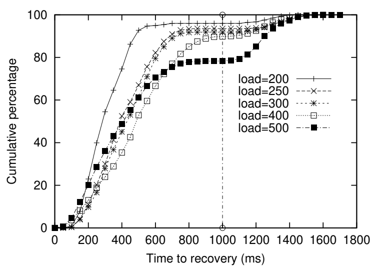
4.2 Time to path recovery: end-to-end recovery

In this subsection, we study the system behavior with an increasing number of simultaneous client sessions, while using the end-to-end recovery mechanism for failed paths. We capture our metric of time-to-recovery of client sessions as a function of the number of client sessions for which an overlay node is the exit node (and hence its CM is responsible for path creation and recovery for that session). In the rest of the discussion, we refer to the number of client sessions for which an overlay node is an exit node as the *load L* on it (or equivalently, the load on its CM).

In this set of experiments, we first use a 20-node overlay network (with 54 edges) generated as described earlier, and study scaling in the dimension of the number of clients. We later consider the effect of increasing the overlay size. There are a total of ten different services, "s0" through "s9", each with two replicas in the overlay network. Having two replicas ensures an alternate server for failure recovery, and having ten different kinds of services with two replicas each ensures that the overlay network is uniformly covered with services. The replicas are placed at random locations in the overlay. Each client path request involves two different randomly chosen services from among the ten. (Note that although each path has only two logical services, the path could stretch across many more overlay nodes, via the no-op services).

Across the runs, we vary the load *L* from an initial value of 25 paths per CM, and increase it gradually to examine the scaling behavior. We have equal load at all the 20 CMs. For a given load, we first establish all the paths (total #paths = #paths terminating at a CM × 20 CMs). We then deterministically fail the link in the overlay network with the *maximum* number of client sessions traversing it. This is the worst case in a single-link failure. We conclude the experiment shortly after all the failed paths have been recovered (a few seconds). We then compute the time to recovery, averaged over all the paths that failed and were recovered. Fig. 6(a) shows this average metric plotted against the load as we





Load L	#pkts lost by CMs	#pkts lost by emul.	Max rate at emul. (pkts/s)
200	0	99	20,640
250	0	163	19,240
300	0	201	19,630
400	0	578	21,200
500	0	753	21,590

Figure 6. (a) Time to recovery vs. Load (b) CDF of time-to-recovery for different values of load (c) Detecting the bottleneck

defined above. The error bars indicate the standard deviation.

There are several things we note about the plot. Firstly, the average time to recovery remains low, below 600 ms even for a load of up to 500 paths per cluster manager. This time-to-recovery is the time taken for signaling messages to setup the alternate path, after failure detection. Secondly, the average time-to-recovery increases only slowly as the load increases. This suggests that the system has not reached its saturation point yet. That is, even at higher load, queuing delays associated with processing the distributed recovery messages are minimal. The third observation we make is that the variance of the time-to-recovery across all failed paths is large at high load. To explain this, we plot another graph.

Fig. 6(b) shows the CDF of the time-to-recovery of all the failed paths. Different plots are shown for different values of the load. We see that the majority of the paths recover well within 1 sec, and a small fraction of the paths take over 1 sec to recover (notice the flat region in the CDF). This is due to the following reason. The path recovery control messages are transmitted using the reliable UDP messaging layer of Fig. 4. This layer implements a re-transmit after 1 sec, if there is no reply to the first packet[1]. Such a re-transmit occurs for the path recovery control messages since the first control message is lost, at higher load. A certain fraction of the paths being recovered thus experience significantly higher recovery time than others. This explains the high variance at high load, in Fig. 6(a).

There are two reasons why packet losses can occur: (1) excess load in processing the path recovery messages at the CMs, or (2) bottleneck at the emulator in our setup. (Note that we have not yet modeled packet-losses/outages on the overlay links. Also, the control packet losses could not be because of the deterministically failed link, since our algorithm does not send any recovery messages on the failed link itself). Case (1) would mean that we have a bottleneck in our software architecture, while case (2) would mean that the emulator setup is being stressed. To check this, we instrument the emulator to: (a) count the number of packets it sent and received, and (b) measure the packet rate it saw, in 100 ms windows. The CMs also keep track of the number of packets they send and receive. Using (a), we compute the number of control packets lost at the CM, and the number of control packets lost in the emulator setup. We use (b) to check against the emulator limits given in Table 1 of Section 3.

In Fig. 6(c), we tabulate these values for different loads. We notice that there are no packet losses at any of the CMs, meaning that the bottleneck is not in the message processing at these nodes. However, the emulator node (or the local area network in-between) loses a small number of packets, and this number increases with the load in the system. The table also gives the maximum rate seen by the emulator in 100 ms windows. Referring back to Table 1, we see that the emulator setup is close to its limits in these experiments, in terms of the packet rate. (The sizes of all control packets were within 300 bytes). Note that for every packet lost by the emulator, a client session recovery could experience a control message re-transmit, and thus a recovery time higher than 1.0 sec.

We thus conclude with certainty from the above experiments that the system can handle at least 200 paths/CM easily. Also, since *no* packets are lost by the CMs due to processing bottlenecks (column 1 of the table in Fig. 6) even at higher loads, we can say with reasonable certainty that the scaling limits of the CMs have not been reached even at loads of 400-500 paths/CM. This is also corroborated by the fact that the average time-to-recovery increases only slowly with increasing load – if saturation point had been reached, we would have expected to see a steep increase in the plot at this saturation point.

Our cluster manager machines are Pentium III 500MHz quad-processor machines. During our experiments, since the cluster was in production use, we were not able to get fully unloaded machines, but always used the least loaded set of machines. The number of 400-500 simultaneous paths per cluster manager is a reasonable number, since we are dealing with heavy-weight application services such as video transcoders, text-to-speech converters in our examples given earlier. For comparison, the text-to-speech service we implemented in [12] could support only about 15 simultaneous client sessions on hardware similar to those running our CMs. This means that in deploying a service cluster, the amount of provisioning required for cluster manager functionality would be small in comparison to that required for actual services such as the text-to-speech engine. Also, note that a cluster can have multiple CMs dealing with different sets of client path sessions – the system can be provisioned with more cluster managers to support a larger number of simultaneous client sessions.

We make another observation. We have used latency as a metric for path creation, and in the above experiments, failed the overlay link with the *maximum* number of client paths traversing it. This represents a worst-case scenario. This is because, as is well known, a metric such as latency is very poor in distributing load across the network. In fact, in our experiments above, we observed that the load across the overlay nodes was highly skewed. The system can be expected to scale even better if a load balancing metric such as cluster-load is used. We have implemented such a load balancing metric, described in [8].

In the above experiments, we have not considered scaling along the dimension of the number of overlay nodes. However, intuitively, if we grow the overlay network size, and correspondingly

[1] We use a value of 1 sec for the first re-transmit, 1.5 sec for the second re-transmit, 2 sec for all further re-transmits.

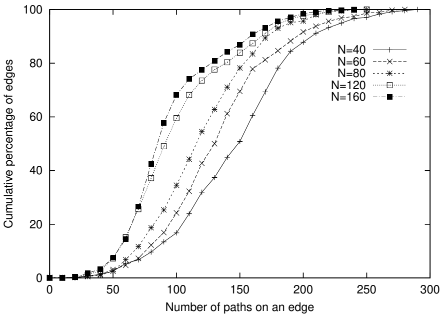


Figure 7. CDF of edge loads, for various overlay sizes

also increase the number of service replicas, the load on the overlay links should remain the same irrespective of the size of the network. In fact, we do observe this experimentally. We generate overlay topologies of various sizes, as explained in Sec. 4.1. We choose a number of service replicas in the overlay proportional to its size. We place these replicas at random locations in the overlay. We setup a number of client sessions, and then measure the load on each overlay link. We plot a CDF of this edge load across all the edges in the overlay, for different overlay sizes. Fig. 7 shows this set of plots. We see that as the network size grows, the load distribution across the various edges does not change much. In fact, with greater connectivity for the larger networks, the edge load only evens out. This is suggested by the fact that the CDF becomes more vertical at the middle with increasing overlay size. Even the maximum edge load does not change with growing overlay size. These observations mean that a link failure in a larger network, with a proportionally larger number of clients, is no worse than in a smaller network. That is, scaling with respect to the number of clients does not worsen with increasing overlay size.

4.3 Time to path recovery: local recovery

We now examine the alternate method of *local* recovery where the failed edge is replaced by a local path. This recovery mechanism has the advantage over end-to-end recovery that since the signaling messages are local, the recovery time can be lower. However, since the path is being fixed locally, we might lose out on global optimization. That is, the resultant path after local recovery might have a higher cost than if end-to-end recovery had been used. We look at the nature of this trade-off now.

Like in our earlier set of experiments, we have a set of runs with varying load; in each run, we create paths before-hand, and then fail the overlay link with the maximum number of paths going through it. Apart from the trade-off mentioned above, there is a further issue with local recovery. Since paths are constrained to pass through nodes with services, they may not be simple graph paths: they may have repeated occurrences of nodes or edges in them. An example is shown in Fig. 8(a). Since local recovery hides the recovery information from the rest of the nodes in the path, handling race conditions in distributed messaging, when there are multiple occurrences of nodes in the original path, becomes difficult. For this reason, we fall back on end-to-end recovery when the original path has repeated occurrences of nodes.

Hence in each run, we use local recovery for client sessions whose original paths do not have repeated nodes, and end-to-end recovery for other client sessions. In each run, there were a significant fraction (at least 25%) of client sessions in each category – it was not the case that one kind of recovery was applied for most client sessions in any run. This has the side effect of making our comparison simpler, since we can compare the average time-to-recovery of paths, under either algorithm, in the same run. The two plots in Fig. 8 illustrate the trade-off between the two algorithms. The first graph shows the average time-to-recovery as a function of the load, much as in Fig. 6(a). The second graph shows the other metric: the ratio of the cost of the recovery path, to the cost of the original path, as a function of the load. (Recall that the path cost in our case the end-to-end latency).

In the first graph, we note that the time-to-recovery has low values, around 700 ms, as earlier. Also, the variance in the time-to-recovery goes up with load, as in Fig. 6(a). The small non-uniformity in the plot is understandable given the magnitude of the variance. Another point we note is that local recovery has consistently lower average recovery time, as expected. Although it has lower time-to-recovery, we note that the difference is very low in absolute terms – within 200-300 ms. (As our discussion in Sec. 4.4 will show, these small differences will be dwarfed by the time to failure *detection* in Internet paths – about 1.8 sec).

The second graph shows the flip side of local recovery – it results in paths that are costlier than with end-to-end recovery. Here, the difference between local and end-to-end recovery are significant. Local recovery results in paths that are 20-40% costlier than the original path, due to the additional re-route in the middle of the original path. On the other hand, end-to-end recovery causes a maximum extra cost of 10% over the original path, and in many cases actually improves the path cost over the original path. Improvement in path cost over the original path is due to the following reason. The latency metric along overlay links is variable, as explained in Sec. 4.1. Hence the original min-cost path is no more the min-cost path after a while – at the time of path recovery. Hence, when an alternate end-to-end path is setup, it can incur a lower cost than the original path. While these differences of 10-30% one way or another may not greatly affect the performance of the client path when using the latency metric, it is significant if we use a graph metric such as load on the cluster node.

4.4 Internet failure behavior and Failure detection

So far in our experiments, the failures in the overlay links have been artificially introduced. We have not modeled how often Internet path failures happen, or how long they last. While this allowed us control over our experiments to understand the system behavior, we would like to see our system performance given Internet path failure patterns. Further, an aspect we have not addressed so far is, how quickly failures can be detected, reliably. We turn to these issues now.

Failure detection: A key aspect of our system is its ability to detect failures in Internet paths. To achieve high-availability, we need to detect failures quickly. In particular, we are concerned about keeping track of the liveness of the wide-area Internet path between successive components in the service-level path. An example is shown in Fig. 3 – the first leg of the service-level path. This is important since unlike the telephone network, the Internet paths are known to have much lesser availability [1, 2].

While the notion of failure if very application specific, for our purposes, we consider Internet path outages such as those that

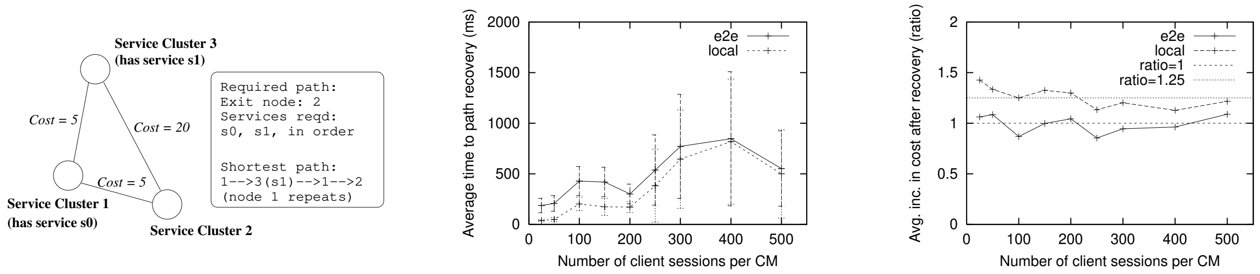


Figure 8. (a) Node repetition: an example (b) Local vs. E2E recovery (time-to-recovery) (c) Local vs. E2E recovery (path cost)

happen when there is a BGP-level failure [2]. These could last for several tens of seconds to several minutes. We wish to detect such long outages. In the rest of the discussion, we use the terms "failure" and "long-outage" interchangeably, and both refer to instances when no packet gets through from one end of the Internet path to the other for a long duration such as several tens of seconds.

The straightforward way to monitor for liveness of the network path between two Internet hosts is to use a keep-alive heart-beat, and a *timeout* at the receiving end of the heart-beat to conclude failure. This is shown in Fig. 9. There is a notion of a *false-positive* when the receiver concludes failure too soon, when the outage is actually not long-lasting (Fig. 9(c)). False-positives occur due to intermittent congestion/loss. We term a path restoration triggered by such a false-positive to be a *spurious* path restoration.

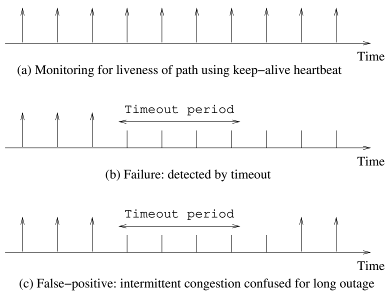


Figure 9. Failure detection using heart-beats

There are three questions to answer in this context. (1) What should be the heart-beat period? (2) Given a heart-beat period, what should the timeout be to conclude long outages? (3) Given a timeout period, how often do false-positives occur, when we confuse intermittent congestion for a long outage? Intuitively, there is a trade-off between the time to failure detection and the rate of false-positives. If the timeout is too small, failures are detected rapidly, but the false-positives increase, and vice-versa when the timeout is too large. We study these in detail now, using wide-area trace data.

Trace data: To answer the three questions posed above, we need a frequency/probability distribution of the incidence and duration of failures. There have been studies of outages or packet loss patterns at small time scales (less than 1 sec) [13, 14]. These have shown that there is correlation of packet loss behavior within one second, but little correlation over a second. Further studies have estimated failures that last for over 30 sec [15, 16]. To the best of our knowledge, there does not exist publicly available data, or a study, that gives a probability distribution of these failure gap periods on a wide-area Internet path.

We have collected data to arrive at such a probability distribution. We run a simple UDP-based periodic heartbeat between pairs of geographically distributed hosts. We choose a heart-beat period of 300 ms. This is based on the fact that packet losses are highly correlated within 1 sec [13, 14]. The set of hosts from which we collected data are: Berkeley, Stanford, CMU, UIUC, UNSW (Australia), and TU-Berlin (Germany). This represents some trans-oceanic links, as well as Internet paths within the continental US (including Internet2 links). We have data for nine pairs of hosts among these, a total of 18 Internet paths. Six of the nine pairs of data were collected in Nov 2000, and three in Oct 2001. One pair of hosts was a repeat between these two runs. Across these eighteen paths, the RTT varied from about 3 ms to 220 ms. The number of AS domains on each Internet path varied between 3 and 6 for these eighteen paths. The heart-beat exchange was done for an extended period of time – for 3-7 days for the 9 pairs of hosts.

To understand the nature of Internet path outages, we compute the gaps between successive heart-beats at the receiving end. Looking across all gap-lengths in an experiment, we get a distribution. Fig. 10(a) shows this distribution as a CDF for 3 pairs of hosts (six Internet paths). The plots for other host-pairs are similar and we do not show them here.

Note that the y-axis in Fig. 10(a) starts from 99.9%. This is because a large number of gaps in reception that are between 300 ms and 600 ms. This is merely inter-arrival jitter since the heart-beat period itself is 300 ms. In the graph, we first draw attention to the last plot which is marked as the ideal case. This is with fictional data, has no connection with our trace data, and is for purposes of illustration. We term this plot as ideal since there is a long flat region in the CDF. This flat region starts from 1,800 ms and continues up to 30,000 ms (30 sec), before the CDF begins to increase again (this increasing part beyond 30 sec cannot be seen on the graph). This long flat region means that, if we choose a timeout of 1,800 ms for detecting long outages, we would never confuse an intermittent congestion for a failure. That is, all intermittent congestion lasts for less than 1,800 ms. An outage lasting 1,800 ms implies an outage lasting for longer than 30,000 ms.

We observe that the plots with the real data are very close to the ideal case. There is a sharp knee in the plot, and the CDF has a region that is almost flat beyond this knee. This suggests a value for the timeout (for failure detection) that is just beyond the knee in the plot. For the different plots, this value varies between 1,200 ms and 1,800 ms.

The region beyond the knee is "almost" flat, but it definitely has

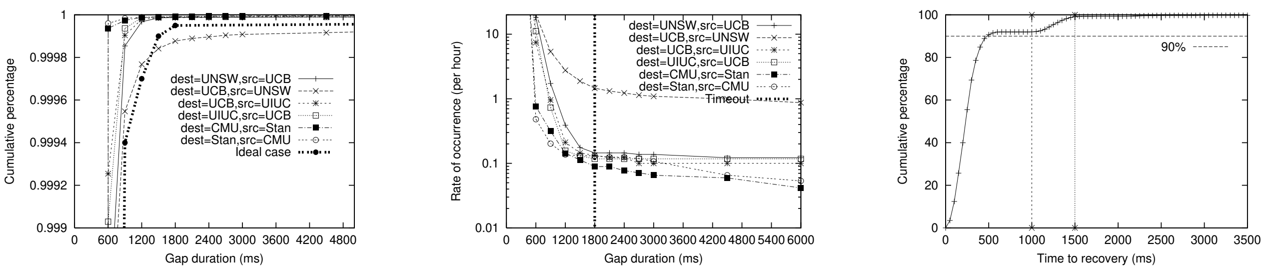


Figure 10. (a) Gap distribution (CDF) (b) Outage occurrence rate (c) Performance under Internet failures

a small slope in all the plots with the real data. This slope means that there is a non-zero probability that we confuse intermittent congestion with long outages or failures. That is, there is intermittent congestion that lasts for periods of time ranging beyond the timeout value. To give an quantitative idea of this observation, suppose that we want to detect outages lasting for 30,000 ms or more, and have a timeout of 1,800 ms. For four of the eighteen Internet paths, the timeout would be able to predict a long outage with probability 40% or more. For six other cases, this probability would be between 15% and 40%, and for the rest of the eight cases, the prediction probability would be less than 15%.

In terms of the prediction probability of long outages, these numbers of 15% or even 40% may not seem good. But we argue that this does not really matter in absolute terms. For this, we plot a second set of graphs in Fig. 10(b). This shows the rate of occurrence of outages of various durations on a log scale. If we consider outages of 1,800 ms or above, these occur about once an hour or less frequently. (The long outages happen even less frequently, but contribute significantly to loss of availability). Intuitively, this is a very small absolute rate of occurrence of timeouts, and hence a small rate of occurrence of false positives. (Sec. 4.6 will quantify as to why this rate of occurrence is manageable even under scale, with a large overlay network).

The plots in Fig. 10(b) also have knees at around the same points as those in Fig. 10(a). This also explains why a timeout just beyond the knee (in either of these plots) is appropriate. A value before the knee, say 1,000 ms, for the timeout would mean that timeouts occur much more frequently – 1-2 orders of magnitude more frequently. This in turn implies a correspondingly large absolute rate of occurrence of false positives. On the other hand, a timeout value much beyond the knee, say 3,000 ms does not bring much reduction in terms of rate of occurrence of false positives, but only increases the failure detection time significantly.

We use this set of data in two ways: (1) we use the plots in Fig. 10(a) to model the distribution of outage periods on the overlay links, which are Internet paths, and (2) we use the empirical value of 1.8 sec, as suggested by the knee-points in either set of plots, as a timeout to conclude failures. (In reality, the timeout can be selected dynamically, using the data collected with the heartbeats). We now return our discussion of path recovery time, but with the above modeling of Internet failures.

4.5 Performance under Internet failure behavior

In this experiment, we wish to study two things: (a) the extent of spurious path restorations under Internet outage patterns, and (b) the performance of our recovery messaging under Internet packet losses as given by our traces in the prior section. Given the set of CDFs of outage durations in the earlier section, we fail links in our overlay with a particular probability, for a particular duration, according to the distribution as in Fig. 10(a). For an overlay link in the testbed, we choose one of the 18 distributions at random. We have a fixed timeout of 1.8 sec to detect failures between a pair of overlay nodes. We now run the same experiment, with the 20-node graph, with a load of 300 paths per cluster manager (total number of paths in the system = $20 \times 300 = 6000$). We use only the end-to-end recovery algorithm for this run. We let the system run for a period of 15 min.

During the run, across all the 54 edges in the graph, there are 162 outages that last 1 sec or more, of which 32 outages last 1.8 sec or more, and 7 last for 20 sec or more. There are 11,079 end-to-end recovery attempts triggered. This represents an average of about 2 recoveries per client path session during the experimental run. 10,974 (99.05%) of these recovery attempts were successful.

For a number of the shorter outages, the outage time itself is comparable to the recovery time. Such short outages are, in some sense, false-positives that trigger spurious path restorations. Ideally, these should not have triggered any recovery – but this happens due to our aggressive timeout mechanism to detect failures quickly. To quantify the fraction of spurious path restorations in our experimental run, we count the number of recovery attempts that were a result of a failure lasting less than 3 sec.

We find that, of the 11,079 recovery attempts, 6,557 (59.18%) are caused by such short outages. This figure of about 60% for the fraction of spurious restorations triggered merits some discussion. We first note that even if a recovery attempt is spurious, application data is not lost any more than during normal Internet performance, without our recovery algorithms. This is because the original path is torn down only after the new path has been established. The only overhead of a spurious recovery attempt is in the control messages introduced by our service composition layer. The control overhead itself is minimal, and can easily be handled with little additional provisioning in terms of cluster managers, as shown in Sec. 4.2. In absolute terms, spurious path restorations and failures themselves occur infrequently. The average rate of occurrence of failures per link in our experimental run is: $\#outages\ over\ 1.8\ sec/\#links/15\ min = 32/(54 \times 2)/0.25 = 1.2/hour/link$. The rate of occurrence of spurious restorations is even lower since only a fraction of the outages represent spurious failure detections. Hence spurious restorations are a small price to pay for the benefits of quick failure detection with an aggressive timeout.

An important aspect of path restorations (including spurious ones) is that of system *stability*. If the absolute rate of occurrence of path restorations is high in the system, instability could result. That is, paths could be switched repeatedly, with cascading or alternating failures due to overload in portions of the overlay network. In our experiment above, we did not observe any such instability. In retrospect, the reason for this is simple – our system can easily handle loads of 300 paths/CM (which is what we had in the experiment above), and there are no processing bottlenecks that drive the system to an unstable state.

Fig. 10(c) shows the CDF of the time-to-recovery of all the paths. Note the flat region in the CDF, as in Fig. 6(b). This represents a re-transmit of a control message during path recovery. Such re-transmits are due to the Internet packet losses we have modeled in this experiment. The plot indicates that over 90% of the recoveries are completed within 1 sec. This represents the recovery time under packet loss as modeled by our outage periods. Such a quick restoration represents orders of magnitude better performance than Internet path recovery that takes several tens of seconds to minutes [2].

4.6 Other sources of overhead

So far we have focused on the path recovery algorithm component of our architecture. The other pieces are (1) the peer-peer heart-beat and measurements, (2) the link-state propagation, and (3) the path creation algorithm itself. The first consumes minimal resources: the heart-beat is sent every 300 ms in our implementation. And peer-peer latency measurements are done once every 2 sec. The bandwidth consumed by these is minuscule.

The second, link-state propagation, is performed whenever there is a change in the link-status (dead/live), or when there is a significant change in the latency over the link. Apart from this, we also have a soft-state link-state propagation every 60 sec to handle dynamic graph partitions. Given the nature of latency variation as described earlier, sudden large changes in latency are rare. So most link-state floods are sent over the network due to link failures or restorations. In the experiment we described in the previous subsection, 150 link-state floods happen over the entire run of the experiment lasting 15 min, notifying nodes of a link failure or link recovery. Given that a link-state flood means a single message over each link in the graph, there are only 150 messages per link due to these floods over the entire run. This is also minimal. We expect this number to increase linearly as the number of edges in the graph increases. This is not too bad however, since we do not stipulate a complete graph for the overlay network as in [4]. In fact, it is ideal if the overlay network maps onto the underlying physical network closely. That is, it is good if multiple overlay links do not share underlying physical links.

Another possible source of overhead is the graph computation involved during path creation and path recovery. The complexity of Dijkstra's algorithm is $E \times log(N)$, where E is the number of edges and N is the number of nodes in the graph. In Section 2.1 we mentioned that the algorithm is applied on a transformation of the overlay graph. In turns out that this transformation does not affect the algorithm complexity. In our implementation, this algorithm performs quite well. We performed micro-benchmark studies (not an emulation run) of this algorithm alone, with a 6,510-node overlay network, with 20,649 edges. On the configuration of our cluster machines, the computation takes about 50 ms, and only about 3MB of memory. This figure of 50 ms could be significant overhead if this computation is done for every path creation or recovery. However, we perform an optimization that we term *path caching*. We run the algorithm, and store the resulting "tree" structure for requests for path creation/recovery in the near future. We store one such tree for every kind of service-level path (not every client path session). We update this tree only when the graph state changes – i.e., only 150 times, once for each link-state update, during our experimental run in the previous sub-section. Since we do not run this algorithm for every path creation/recovery, this is not a source of bottleneck.

4.7 Summary of results

In summary, our results show that failure recovery can be performed in our overlay network of service clusters, within 1 sec for over 90% of client sessions (Sec. 4.5). Our trace-data, and the experiments using those show that failure detection can be quite aggressive, with a timeout as low as 1.8 sec, with an infrequent occurrence of spurious path restorations – about once an hour in our experiments. Hence, overall, paths can recover from outages within about $1 + 1.8 = 2.8$ seconds. This would be of tremendous use to applications such as video streaming – without our mechanisms for recovery, client sessions could experience outages that last for several minutes [2]. This figure of 2.8 seconds is definitely good enough for real-time, but non-interactive applications, which usually buffer about 5-10 sec of data. For interactive applications, this may not be perfect, but would provide significantly better end-user experience than without our recovery mechanisms.

Our data shows that there is no bottleneck with the control message processing involved during path recovery, so far as we have been able to scale our emulation testbed. We explored the use of local recovery – while this results in quicker recovery under low load, the local nature of the recovery could lead to sub-optimal path metric for the recovered path.

5 Related Work

The idea of service composition itself is not novel, a simple example being unix piping. The TACC project [6] developed models for fault-tolerance of composed services within a single cluster. The solution is based on monitoring using cluster-managers and front-ends. Apart from the TACC model, cluster-based solutions for fault-tolerance have been studied for other kinds of applications as well. The Active Services [17] model uses a soft-state mechanism for maintenance of long-lived sessions. The LARD approach [18] does load-balancing of client requests for a web-server within a cluster. However, such cluster-based approaches do not address performance or robustness across the wide-area.

In the context of web-servers, the problem of selecting an appropriate service instance based on network and server performance has been studied by earlier approaches [19, 20]. However, for composed services, we have multiple "legs" of the service-level path, and we need to optimize the overall composition, and not just one leg of it. Also, web-server selection mechanisms do not address fail-over for long-lived sessions, since web-sessions typically last for a short period of time (a few seconds).

Routing around failures (above the IP level) in the wide-area has been addressed in other contexts. Content-addressable networks [21] provide an overlay topology for locating and routing toward named objects. The RON project [4] also uses an overlay topology to route around temporary failures at the IP level. In the specific context of video delivery, packet-path diversity has been used as a mechanism to get around failures in [5]. However, these mechanisms are not applicable for composed services – with composed services there is the constraint that the alternate recovery path has to include the component services as well.

The IETF OPES group [3] defines an architecture for "open services" that can be "plugged", or composed. However, this architecture does not include mechanisms for *recovery* when a composed session fails. ALAN [22] proposes application-layer routing by proxylets. The operational model there is different in that the proxylets can be dynamically created and moved around. In our case, the services are deployed by different service providers, and are heavy-weight in nature. Also, ALAN does not have quick-recovery from failures as one of its goals. In our work, we specifically evaluate the recovery aspect of the system.

A unique aspect of our work is the use of an emulation-based testbed for evaluation. Most systems in the networking world are evaluated using either simulations or real experiments – neither of these approaches is suited for our purposes. Our emulation testbed using the Millennium cluster of machines has allowed better modeling than simulations, and more control than real experiments.

6 Conclusions and Future Work

We started with the goal of being able to compose services in a robust fashion, providing recovery mechanisms for long-running client sessions. Our architecture for this is based on an overlay network of service clusters. We have evaluated our architecture for its primary goal of quick recovery of client sessions. Our approach is based on a system distributed across the wide-area Internet. Its evaluation presents a challenge since a simple simulation-based approach would not only have been unrealistic, but would also have failed to identify the bottlenecks in a real system implementation. Developing and maintaining a large scale testbed across the wide-area Internet would have been cumbersome, and would not have been suited for a controlled design study. Our emulation-based approach has allowed a controlled design study with a real implementation. The control overhead in our software architecture is minimal, and requires little additional provisioning. Our trace-driven emulation shows that our recovery algorithms can react within 1 sec. Such quick recovery is possible because we do not have to depend on propagation and stabilization of failure information across the network, to effect recovery. Network failure detection itself can be done with a couple of seconds, with a manageable frequency of spurious path restorations.

There are several avenues for future work, and we point out two possible directions for further exploration in service composition. First, in our work, we have considered only cases of composition where the data flow is a *path*. While this covers a range of useful applications, there are others possible with a more generic definition of composition. Secondly, in our architecture, we have not considered mechanisms for optimizations in other dimensions such as cost to the user. When there are multiple providers they may have different pricing for their service instances. Considering these aspects while making the choice of service instances during recovery pose interesting additional challenges.

Acknowledgments: We thank L. Subramanian, S. Machiraju, A. Costello, S. Agarwal, and the anonymous reviewers for comments on earlier versions of this paper. We thank M. Baker, M. Roussopoulos, J. Mysore, R. Barnes, V. Pranesh, V. Krishnaswamy, H. Karl, Y-S. Chang, S. Ardon, and B. Thai for helping us with the wide-area trace collection.

References

[1] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Wide-Area Network Failures. In *FTCS*, 1999.

[2] C. Labovitz, A. Ahuja, A. Abose, and F. Jahanian. An Experimental Study of Delayed Internet Routing Convergence. In *ACM SIGCOMM*, Aug/Sep 2000.

[3] Open Pluggable Edge Services. http://www.ietf-opes.org/.

[4] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *ACM SOSP*, Oct 2001.

[5] J. G. Apostolopoulos. Reliable video communication over loss packet networks using multiple state encoding and path diversity. In *Visual Comm. and Image Proc.*, Jan 2001.

[6] A. Fox. *A Framework for Separating Server Scalability and Availability from Internet Application Functionality*. PhD thesis, U.C.Berkeley, 1998.

[7] S. Choi, J. Turner, and T. Wolf. Configuring Sessions in Programmable Networks. In *IEEE INFOCOM*, Apr 2001.

[8] B. Raman and R. H. Katz. Load Balancing and Stability Issues in Algorithms for Service Composition. In *IEEE INFOCOM*, Apr 2003.

[9] T. M. Chen and T. H. Oh. Reliable Services in MPLS. *IEEE Communications Magazine*, Dec 1999.

[10] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *IEEE INFOCOM*, Apr 1996.

[11] A. Acharya and J. Saltz. A Study of Internet Round-Trip Delay. Technical Report CS-TR 3736, UMIACS-TR 96-97, University of Maryland, College Park, 1996-97.

[12] B. Raman, R. H. Katz, and A. D. Joseph. Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communication Network. In *WMCSA*, Dec 2000.

[13] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley. Measurement and Modeling of Temporal Dependence in Packet Loss. In *IEEE INFOCOM*, Mar 1999.

[14] J. C. Bolot, H. Crepin, and A. V. Garcia. Analysis of audio packet loss in the Internet. In *NOSSDAV*, Apr 1995.

[15] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the Constancy of Internet Path Properties. In *ACM SIGCOMM Internet Measurement Workshop*, Nov 2001.

[16] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN Service Availability. In *USITS*, Mar 2001.

[17] E. Amir. *An Agent Based Approach to Real-Time Multimedia Transmission over Heterogeneous Environments*. PhD thesis, U.C.Berkeley, 1998.

[18] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *ASPLOS*, Oct 1998.

[19] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *USITS*, Dec 1997.

[20] S. G. Dykes, C. L. Jeffery, and K. A. Robbins. An Empirical Evaluation of Client-side Server Selection Algorithms. In *IEEE INFOCOM*, Mar 2000.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *ACM SIGCOMM*, Aug 2001.

[22] A. Ghosh, M. Fry, and J. Crowcroft. An Architecture for Application Layer Routing. In *IWAN*, Oct 2000.